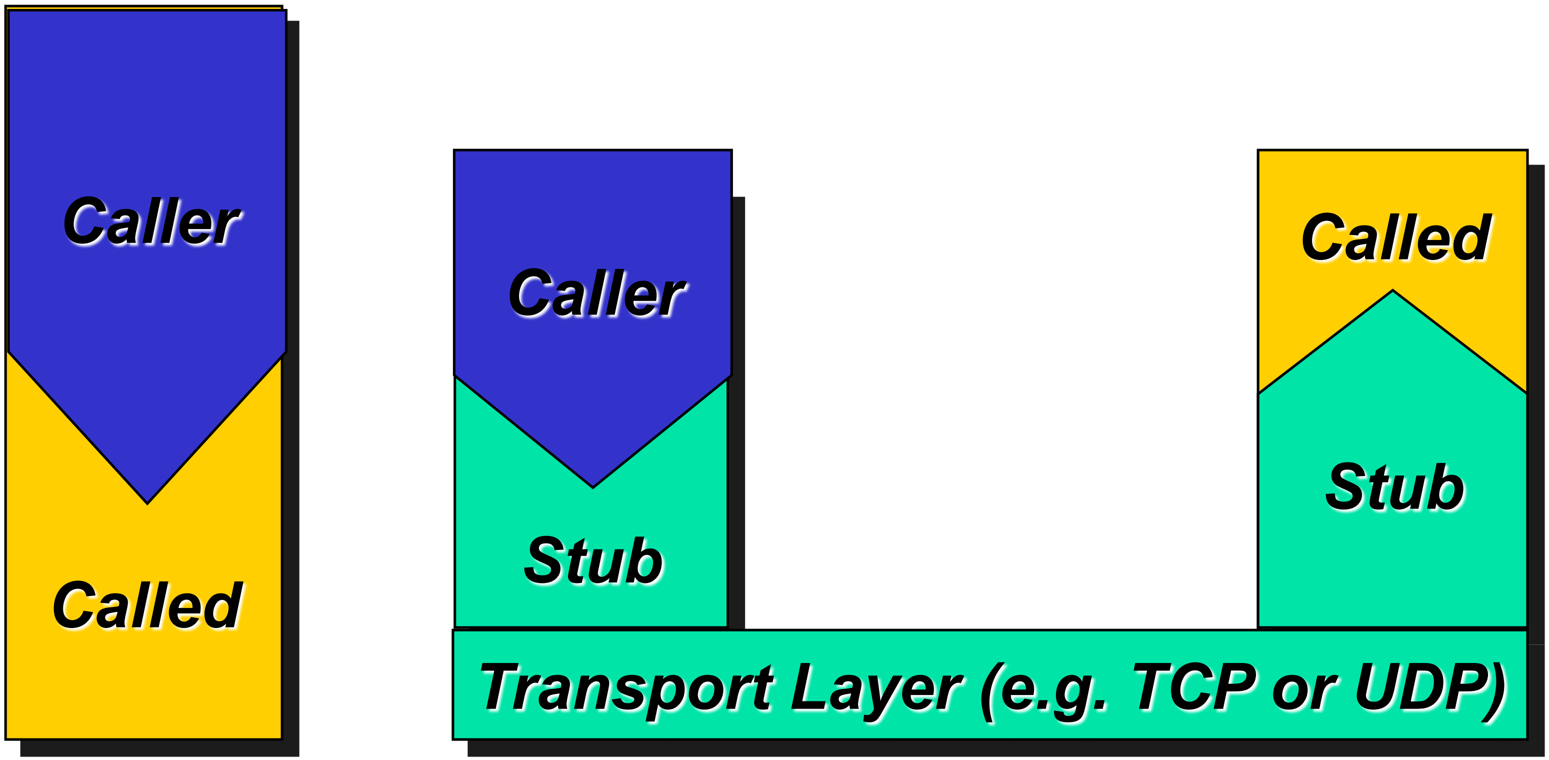
COMP 6231: Distributed System Design

Remote Invocation and RMI

Based on Chapters 5, 7 of the text book and the slides from Prof. M.L. Liu, California Polytechnic State University

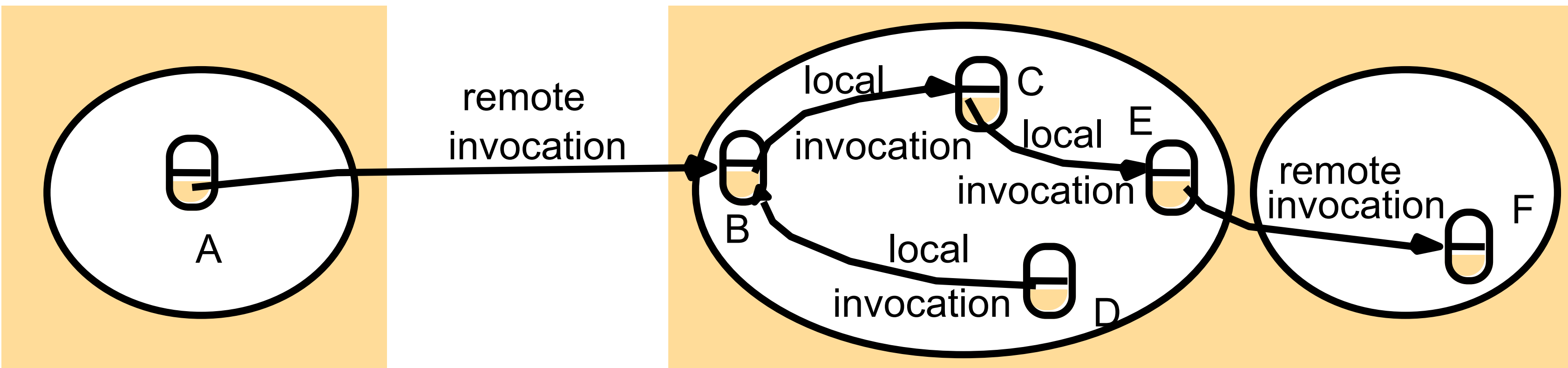
Local versus Remote Procedure Call



Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Remote and local method invocations



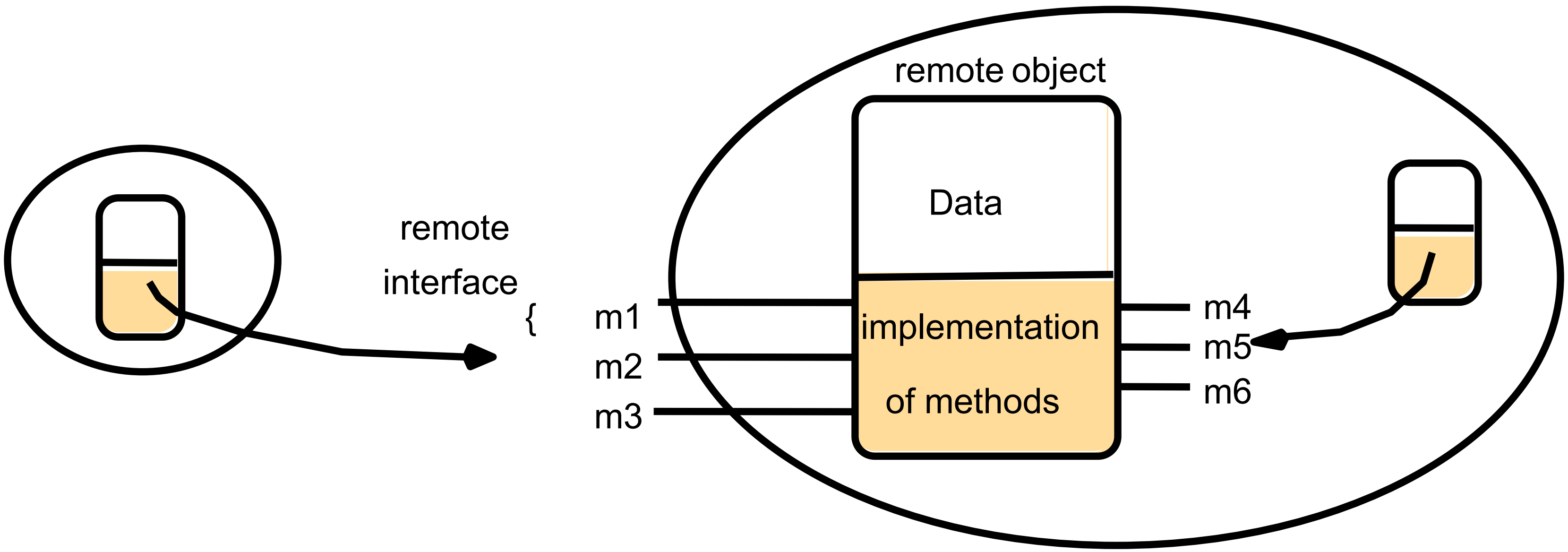
- Each process contains objects, some of which can receive remote invocations, others can receive only local invocations
- Those that can receive remote invocations are called *remote objects*
- Objects need to know the *remote object reference* of an object in another process in order to invoke its methods. How do they get it?
- The *remote interface* specifies which methods can be invoked remotely

Representation of a remote object reference

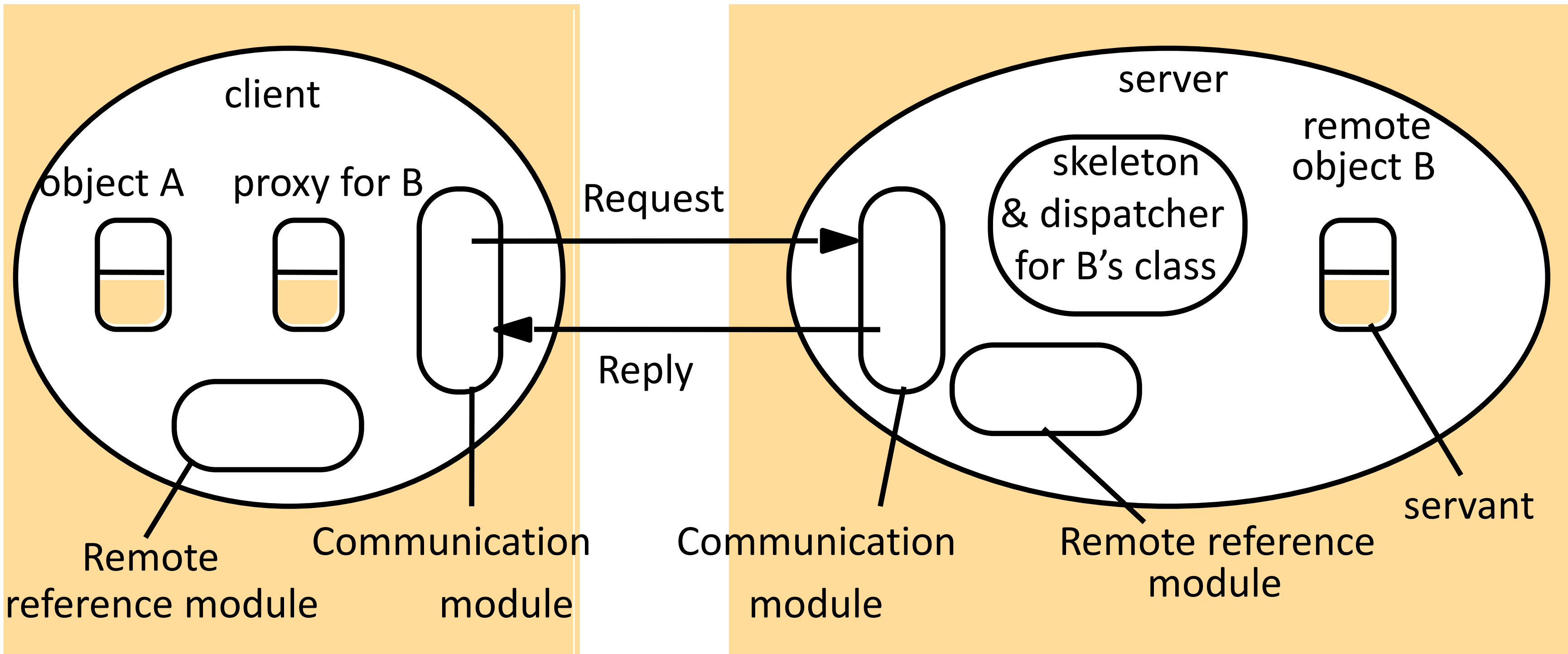
32 bits	32 bits	32 bits	32 bits	
Internet address	port number	time	object number	interface of remote object

- A remote object reference must be unique in the distributed system and over time. It should not be reused after the object is deleted.
- The first two fields locate the object unless migration or re-activation in a new process can happen
- The fourth field identifies the object within the process whose interface tells the receiver what methods it has (e.g. class *Method*)
- A remote object reference is created by a remote reference module when a reference is passed as argument or result to another process
  - It will be stored in the corresponding proxy
  - It will be passed in request messages to identify the remote object whose method is to be invoked

A remote object and its remote interface



The role of proxy and skeleton in remote method invocation



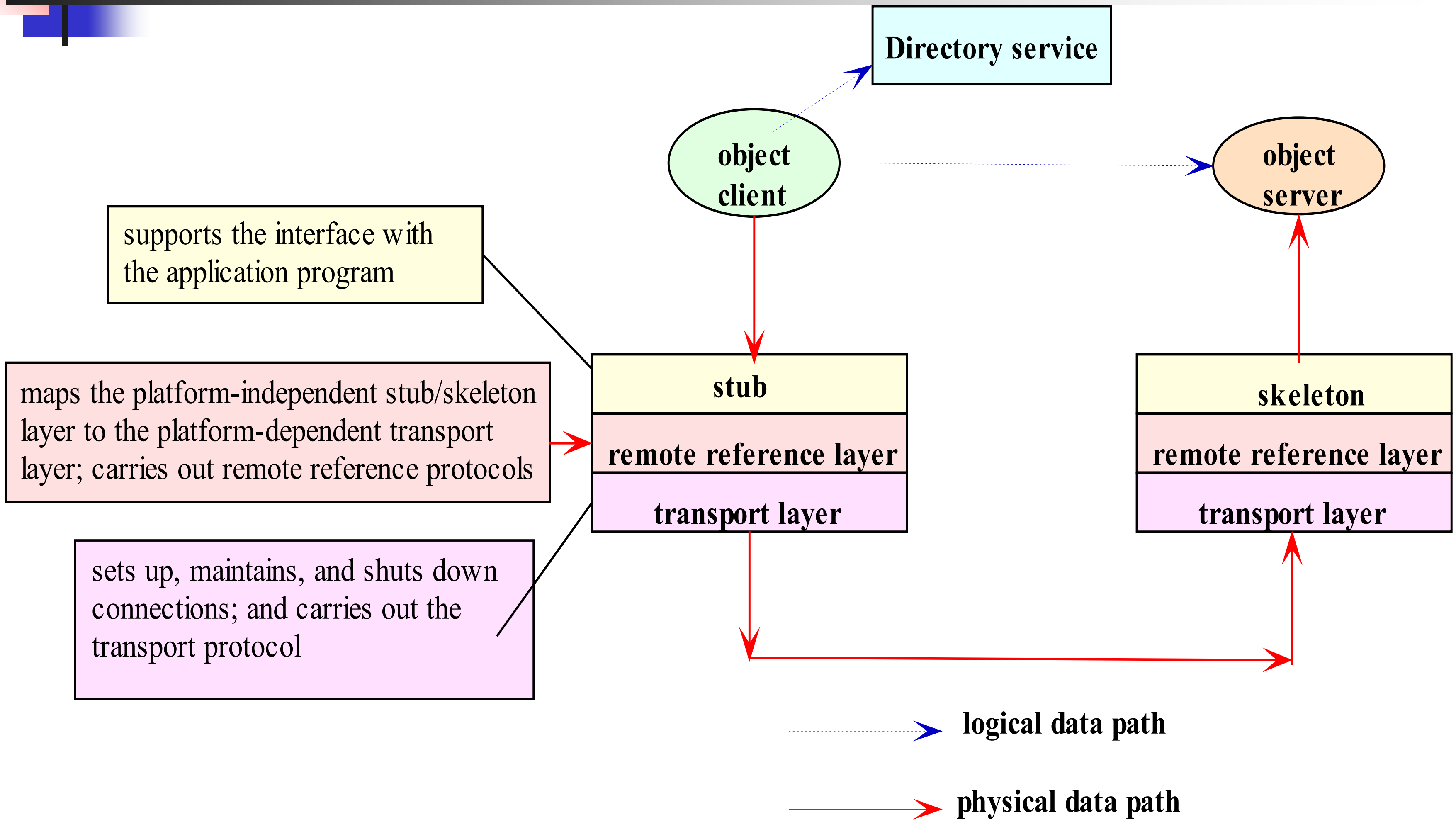
Stubs

- Creating code for marshalling and unmarshalling is tedious and error-prone.
- Code can be generated fully automatically from interface definition.
- Code is embedded in stubs for client and server.
- Client stub represents server for client, Server stub represents client for server.
- Stubs achieve type safety.
- Stubs also perform synchronization.

Synchronization

- Goal: achieve similar synchronization to local method invocation
- Achieved by stubs:
  - Client stub sends request and waits until server finishes
  - Server stub waits for requests and calls server when request arrives

The Java RMI Architecture



Algorithm for developing the server-side software

1. Open a directory for all the files to be generated for this application.
2. Specify the remote-server interface in ***SomeInterface.java***. Compile it until there is no more syntax error.
3. Implement the interface in ***SomeImpl.java*** Compile it until there is no more syntax error.
4. Use the RMI compiler ***rmic*** to process the implementation class and generate the stub file and skelton file for the remote object:

   ```
   rmic SomeImpl
   ```

   The files generated can be found in the directory as ***SomeImpl_Skel.class*** and ***SomeImpl_Stub.class.***

   Steps 3 and 4 must be repeated each time that a change is made to the interface implementation.
5. Create the object server program ***SomeServer.java***. Compile it until there is no more syntax error.
6. Activate the object server : `java SomeServer`

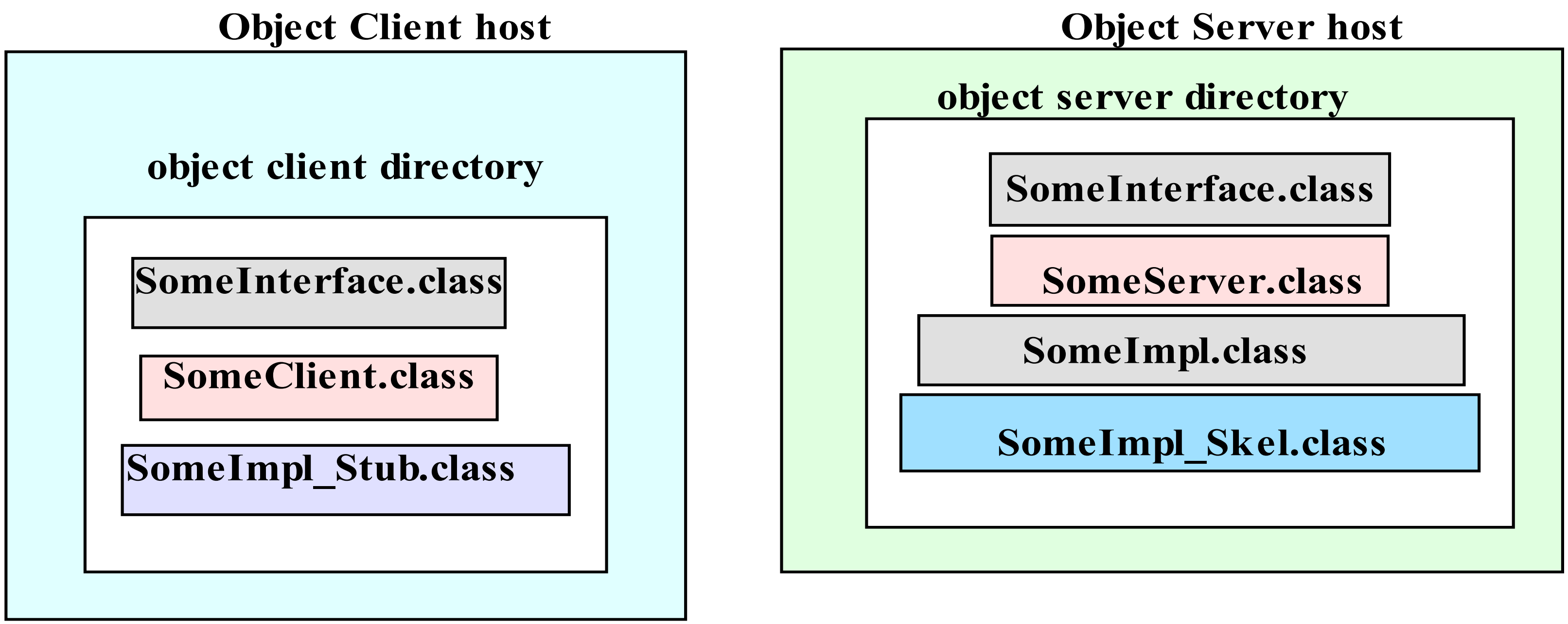
Algorithm for developing the client-side software

1. Open a directory for all the files to be generated for this application.
2. Obtain a copy of the remote interface class file. Alternatively, obtain a copy of the source file for the remote interface, and compile it using ***javac*** to generate the interface class file.
3. Obtain a copy of the stub file for the implementation of the interface:

    SomeImpl_Stub.class.
4. Develop the client program ***SomeClient.java***, and compile it to generate the client class.
5. Activate the client.

```
java SomeClient
```

Placement of files for a RMI application



The *Naming* class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 5.14, line 3.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 5.16 line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the registry.

HelloInterface

```
// A simple RMI interface file - M. Liu
import java.rmi.*;
/**
 * This is a remote interface.
*/
public interface HelloInterface extends Remote {
/**
 * This remote method returns a message.
 *        param  name - a String containing a name
 *        returns a String message.
 */
   public String sayHello(String name)
      throws java.rmi.RemoteException;
} //end interface
```

Implementation of HelloInterface

```
import java.rmi.*;
import java.rmi.server.*;
/* This class implements the remote interface HelloInterface. */

public class HelloImpl extends UnicastRemoteObject
    implements HelloInterface {

   public HelloImpl() throws RemoteException {
      super( );
   }

   public String sayHello(String name) throws RemoteException {
      return "Hello, World!" + name;
   }
} // end class
```

Hello Server

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.net.*;
import java.io.*;
/* This class represents the object server for a distributed object of class
    Hello, which implements the remote interface HelloInterface. */
public class HelloServer  {
   public static void main(String args[]) {
      InputStreamReader is = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(is);
      String portNum, registryURL;
```

Start Hello Server

```
try{
        System.out.println("Enter the RMIregistry port number:");
        portNum = (br.readLine()).trim();
        int RMIPortNum = Integer.parseInt(portNum);
        startRegistry(RMIPortNum);
        HelloImpl exportedObj = new HelloImpl();
        registryURL = "rmi://localhost:" + portNum + "/hello";
        Naming.rebind(registryURL, exportedObj);
        System.out.println("Hello Server ready.");
      }// end try
      catch (Exception re) {
        System.out.println("Exception in HelloServer.main: " + re);
      } // end catch
  } // end main
} // end class
```

Hello Client

```
import java.io.*;
import java.rmi.*;
/* This class represents the object client for a distributed object of class
    Hello, which implements the remote interface HelloInterface.  */

public class HelloClient {
   public static void main(String args[]) {
      try {
         int RMIPort;
         String hostName;
         InputStreamReader is = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(is);
         System.out.println("Enter the RMIRegistry host name:");
         hostName = br.readLine();
         System.out.println("Enter the RMIregistry port number:");
         String portNum = br.readLine();
         RMIPort = Integer.parseInt(portNum);
```

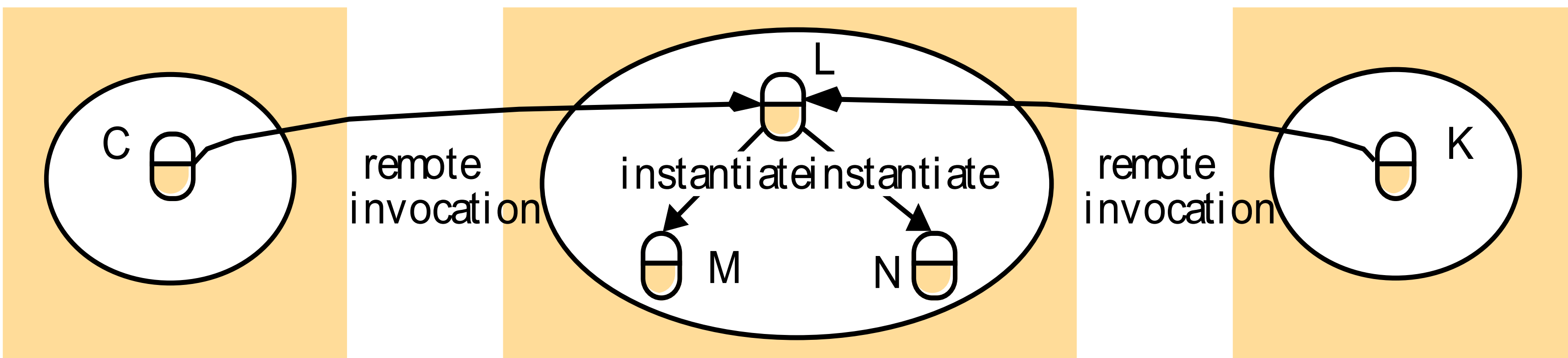
Hello Client, Continued

```
        String registryURL = "rmi://" + hostName+ ":" + portNum + "/hello";
        // find the remote object and cast it to an interface object
        HelloInterface h =  (HelloInterface)Naming.lookup(registryURL);
        System.out.println("Lookup completed " );
        // invoke the remote method
        String message = h.sayHello("Donald Duck");
        System.out.println("HelloClient: " + message);
      } // end try
      catch (Exception e) {
        System.out.println("Exception in HelloClient: " + e);
      }
   } //end main
}//end class
```

Comparison of the RMI and the socket APIs

- The remote method invocation API is an efficient tool for building network applications. It can be used in lieu of the socket API in a network application.
- Some of the tradeoffs between the RMI API and the socket API are as follows:
  - The socket API is closely related to the operating system, and hence has less execution overhead. For applications which require high performance, this may be a consideration.
  - The RMI API provides the abstraction which eases the task of software development. Programs developed with a higher level of abstraction are more comprehensible and hence easier to debug.

Instantiation of remote objects



Multithreading the server

- Three major options:
  - Single-threaded server: only does one thing at a time, uses send/receive system calls and blocks while waiting (*iterative server*)
  - Multi-threaded server: internally concurrent, each request spawns a new thread to handle it (*concurrent server*)
  - Upcalls: event dispatch loop does a procedure call for each incoming event, like for X11 or PC's running Windows.

Single threading: drawbacks

- Applications can deadlock if a request cycle forms: I'm waiting for you and you send me a request, which I can't handle
- Much of system may be idle waiting for replies to pending requests
- Harder to implement RPC protocol itself (need to use a timer interrupt to trigger acks, retransmission, which is awkward)

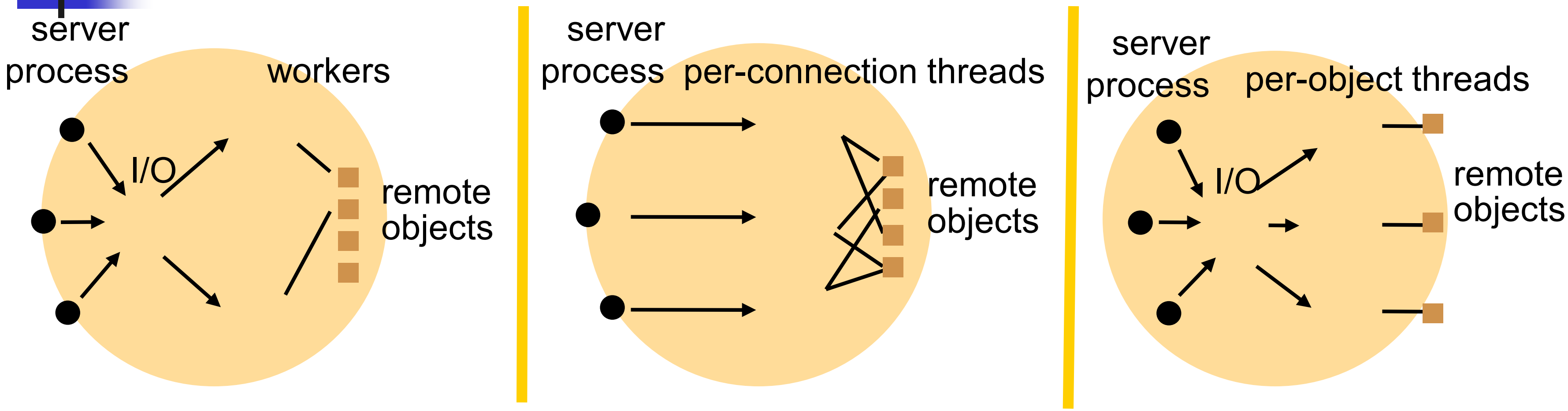
Multithreaded RPC

- Each incoming request is handled by spawning a new thread
- Designer must implement appropriate mutual exclusion to guard against "race conditions" and other concurrency problems
- Ideally, server is more active because it can process new requests while waiting for its own RPC's to complete on other pending requests

Negatives to multithreading

- Users may have little experience with concurrency and will then make mistakes
- Concurrency bugs are very hard to find due to non-reproducible scheduling orders
- Reentrancy can come as an undesired surprise
- Threads need stacks hence consumption of memory can be very high
- Deadlock remains a risk, now associated with concurrency control
- Stacks for threads must be finite and can overflow, corrupting the address space

Server threading architectures



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

- Implemented by the server-side ORB in CORBA
  - (a) Would be useful for UDP-based service, e.g. NTP
  - (b) is the most commonly used - matches the TCP connection model
  - (c) is used where the service is encapsulated as an object. Each object has only one thread, avoiding the need for thread synchronization within objects.

Support for communication and invocation

- The performance of RPC and RMI mechanisms is critical for effective distributed systems.
- Typical times for 'null procedure call':
  - Local procedure call < 1 microseconds
  - Remote procedure call ~ 10 milliseconds
  - 'network time' (involving about 100 bytes transferred, at 100 megabits/sec.) accounts for only .01 millisecond; the remaining delays must be in OS and middleware - latency, not communication time.

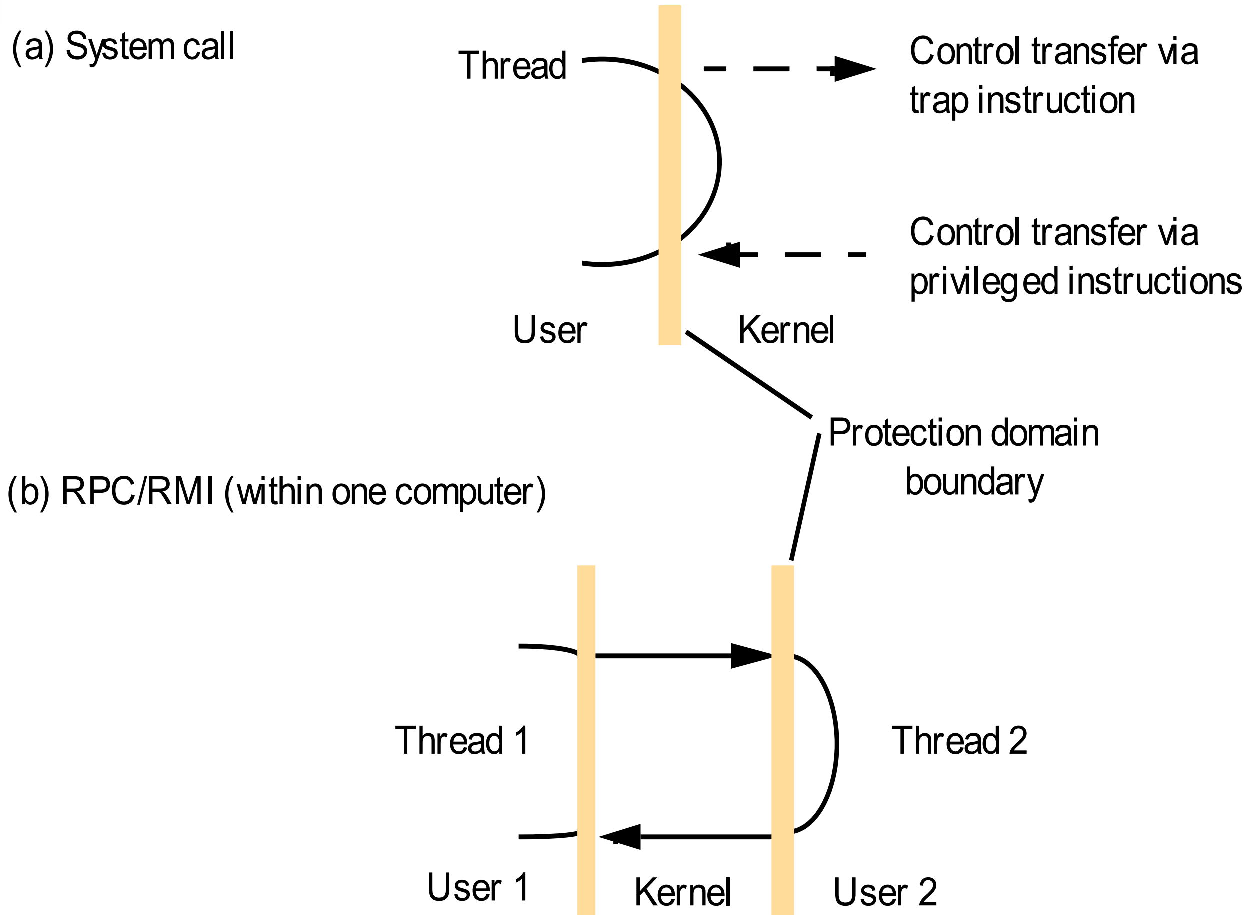
Support for communication and invocation

- Factors affecting RPC/RMI performance
  - marshalling/unmarshalling + operation despatch at the server
  - data copying: application $\rightarrow$ kernel space $\rightarrow$ communication buffers
  - thread scheduling and context switching: including kernel entry
  - protocol processing: for each protocol layer
  - network access delays: connection setup, network latency

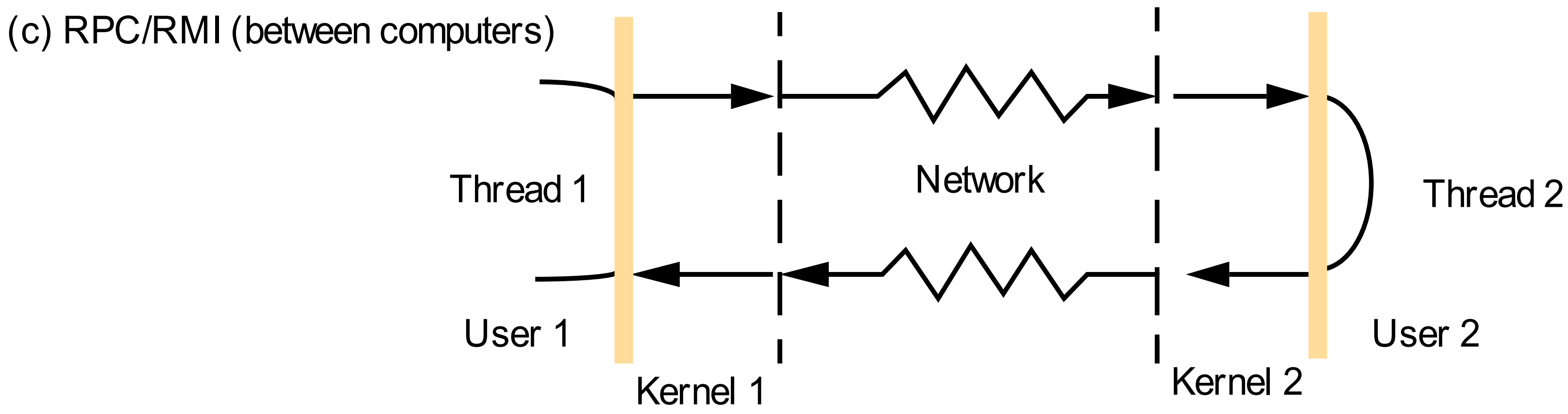
Implementation of invocation mechanisms

- Most invocation middleware (Corba, Java RMI, HTTP) is implemented over TCP
  - For universal availability, unlimited message size and reliable transfer.
  - Sun RPC (used in NFS) is implemented over both UDP and TCP and generally works faster over UDP
- Research-based systems have implemented much more efficient invocation protocols.
- Concurrent and asynchronous invocations
  - middleware or application doesn't block waiting for reply to each invocation

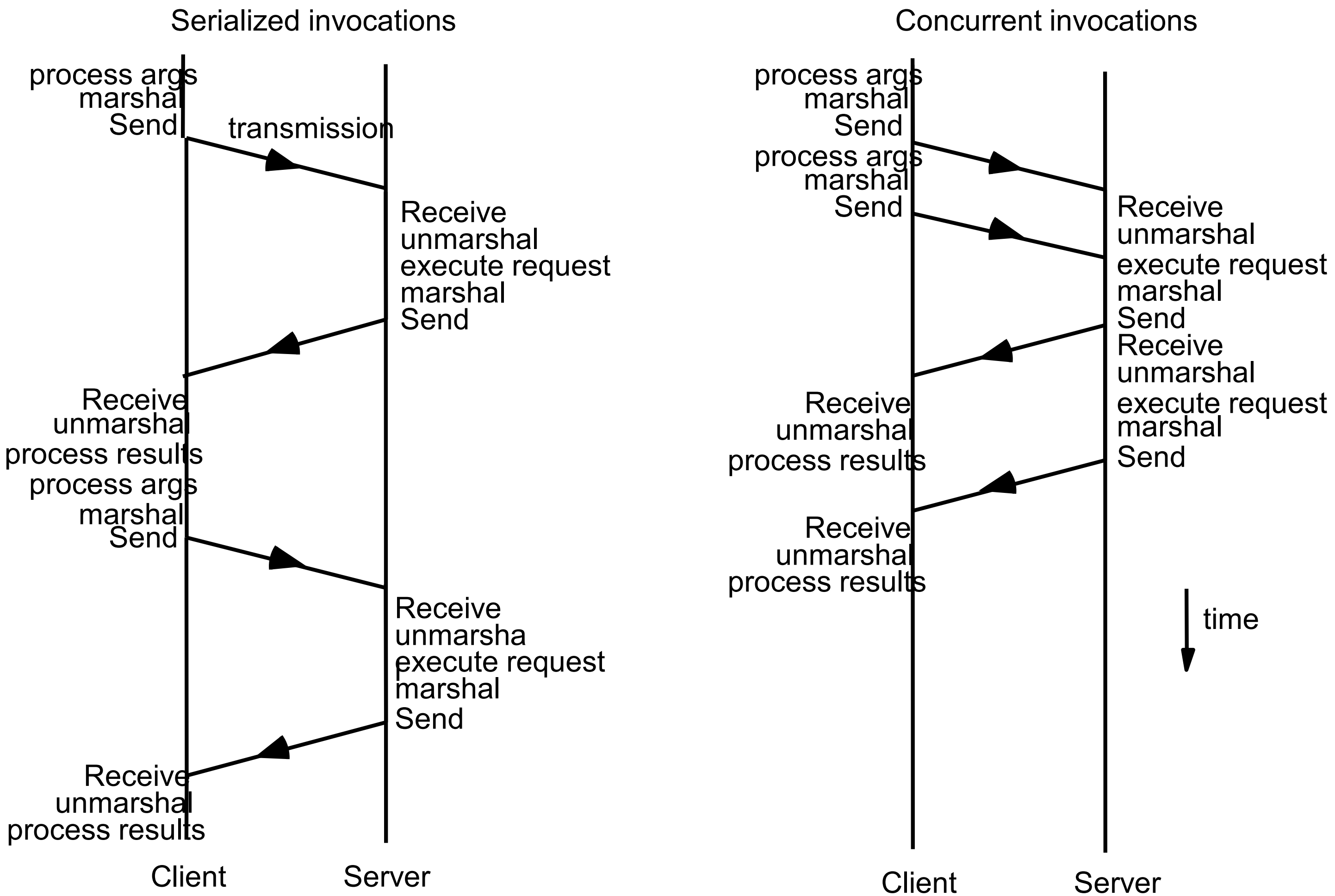
Invocations between address spaces



Invocations between address spaces



Times for serialized and concurrent invocations



Performance: the monster in the closet

- Often, the hidden but huge issue is that we want high performance
  - After all, a slow system costs more to operate and may drive users crazy!
- The issue is that some techniques seem simple and seem to do the trick but are as much as *thousands of times* slower than other alternatives
  - Forcing us to use those alternatives... And perhaps driving us away from what the platforms support

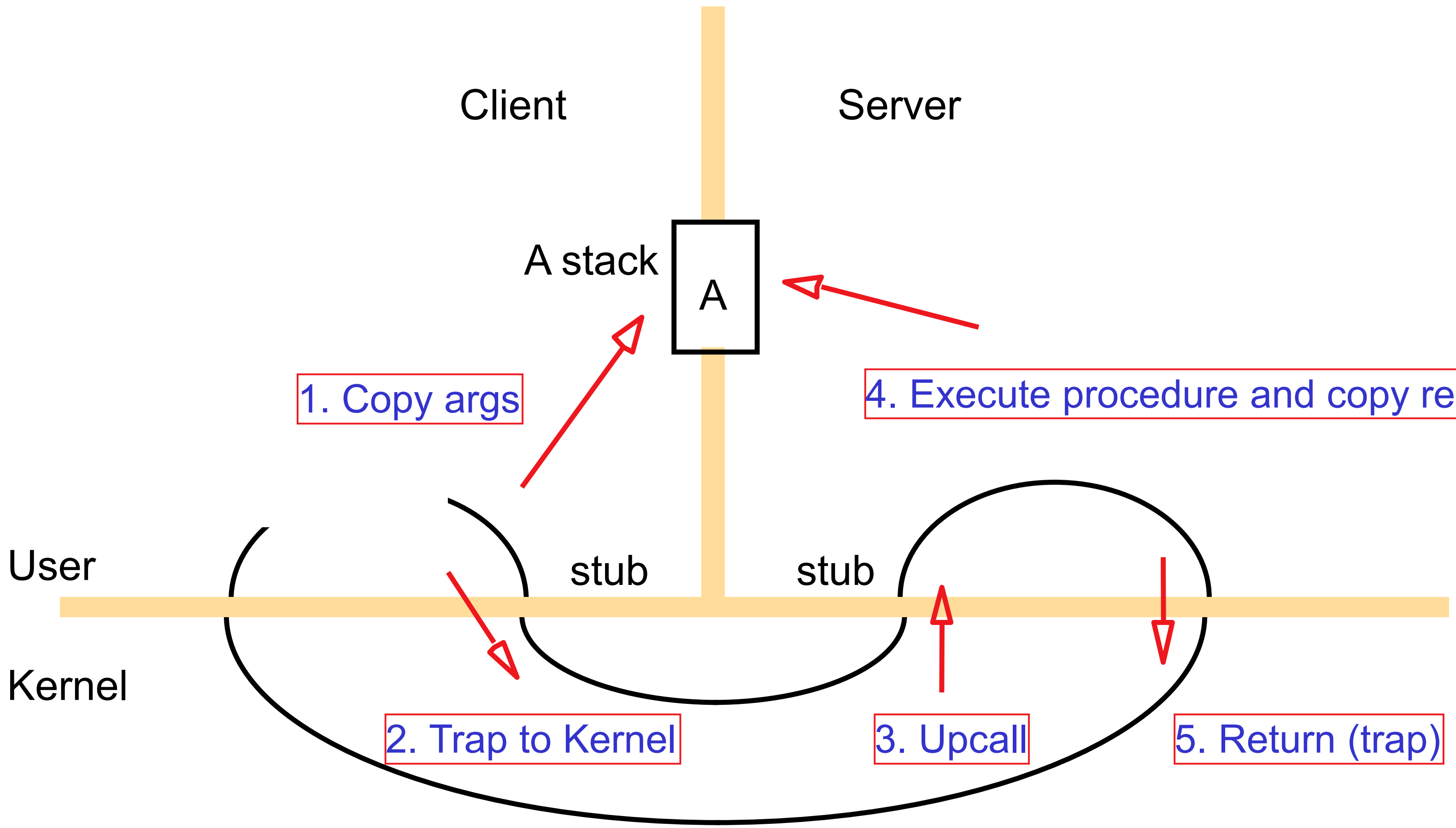
Important optimizations: LRPC

- Lightweight RPC (LRPC): for case of sender, destination on same machine (Bershad et. al.)
- Uses memory mapping to pass data
- Reuses same kernel thread to reduce context switching costs (user suspends and server wakes up on same kernel thread or "stack")
- Single system call: *send_rcv* or *rcv_send*

Bershad's LRPC

- Uses shared memory for interprocess communication
  - while maintaining protection of the two processes
  - arguments copied only once (versus four times for convenitional RPC)
- Client threads can execute server code
  - via protected entry points only (uses capabilities)
- Up to 3 x faster for local invocations

A lightweight remote procedure call

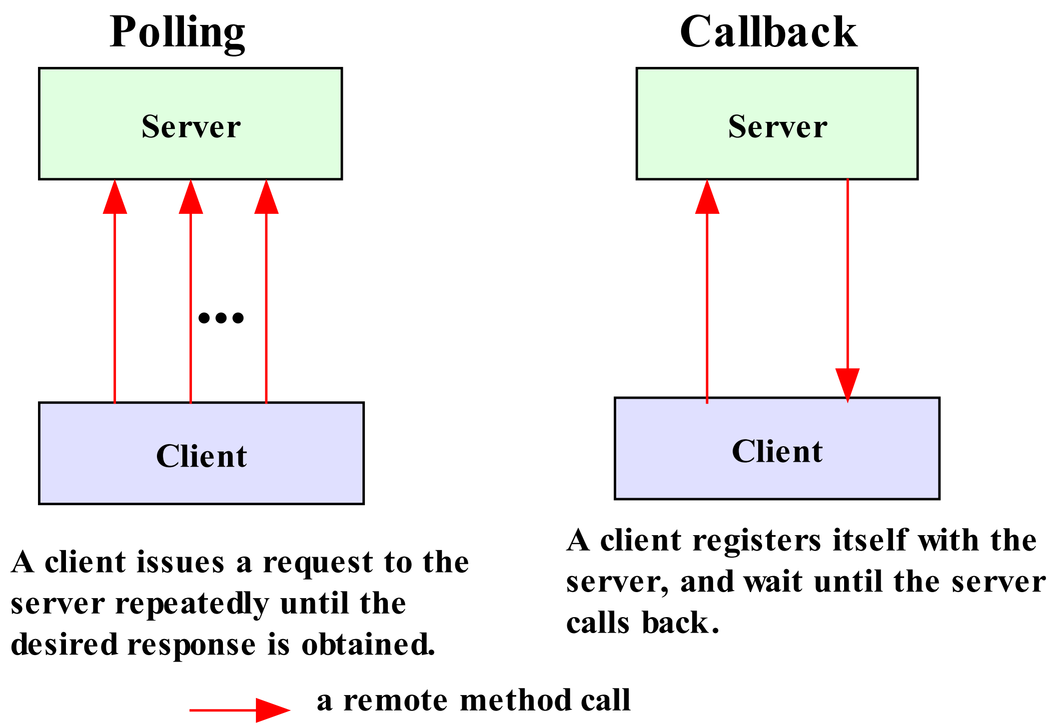


Callback

- In the client server model, the server is passive: the IPC is initiated by the client; the server waits for the arrival of requests and provides responses.
- Some applications require the server to initiate communication upon certain events.
  - monitoring
  - games
  - auctioning
  - voting/polling
  - chat-room
  - message/bulletin board
  - groupware

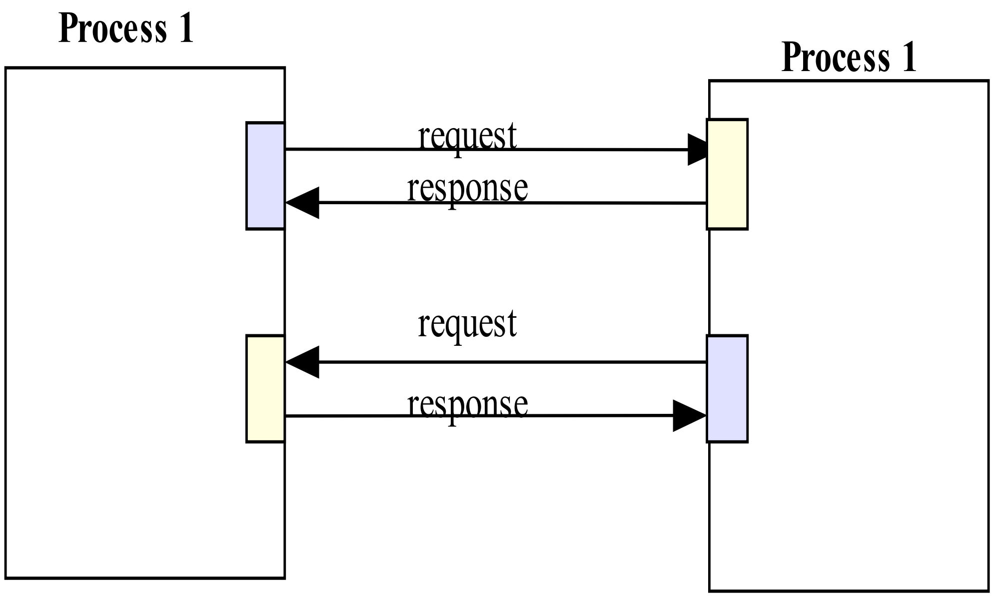
Polling vs. Callback

- In the absence of callback, a client will have to poll a passive server repeatedly if it needs to be notified that an event has occurred at the server end.



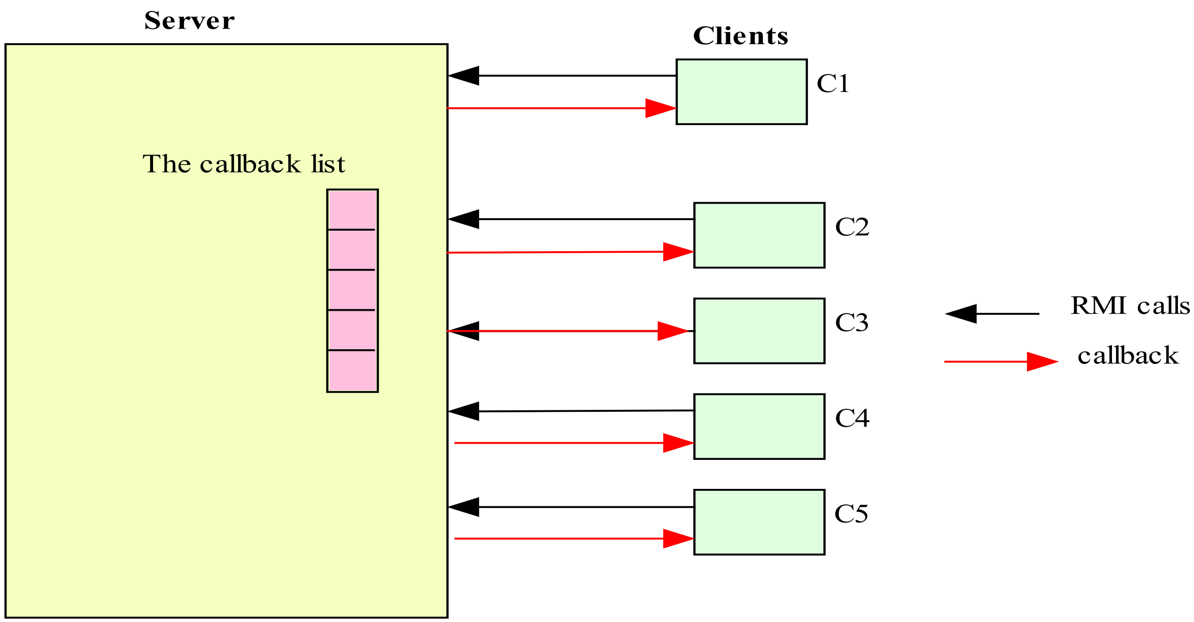
Two-way communications

- Some applications require that both sides may initiate IPC.
- Using sockets, duplex communication can be achieved by using two sockets on either side.
- With connection-oriented sockets, each side acts as both a client and a server.

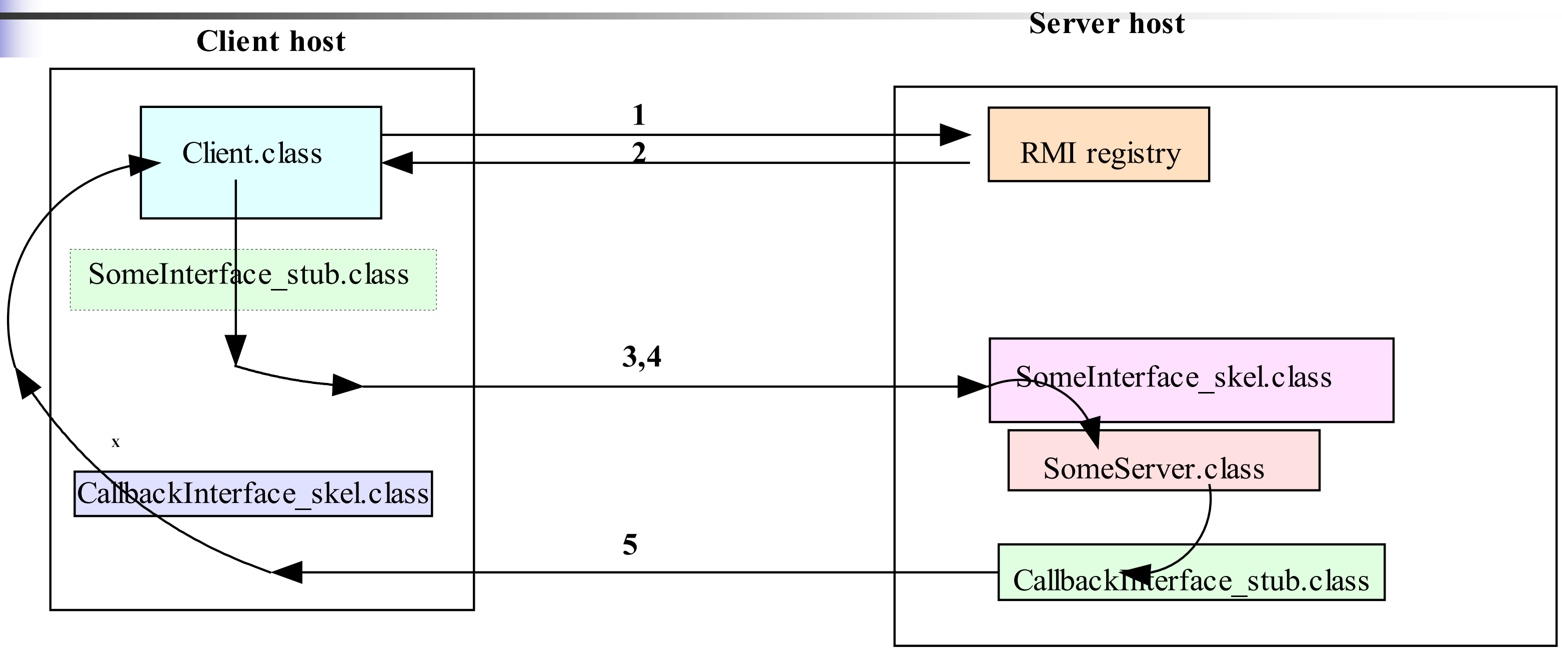


RMI Callbacks

- A callback client registers itself with an RMI server.
- The server makes a callback to each registered client upon the occurrence of a certain event.

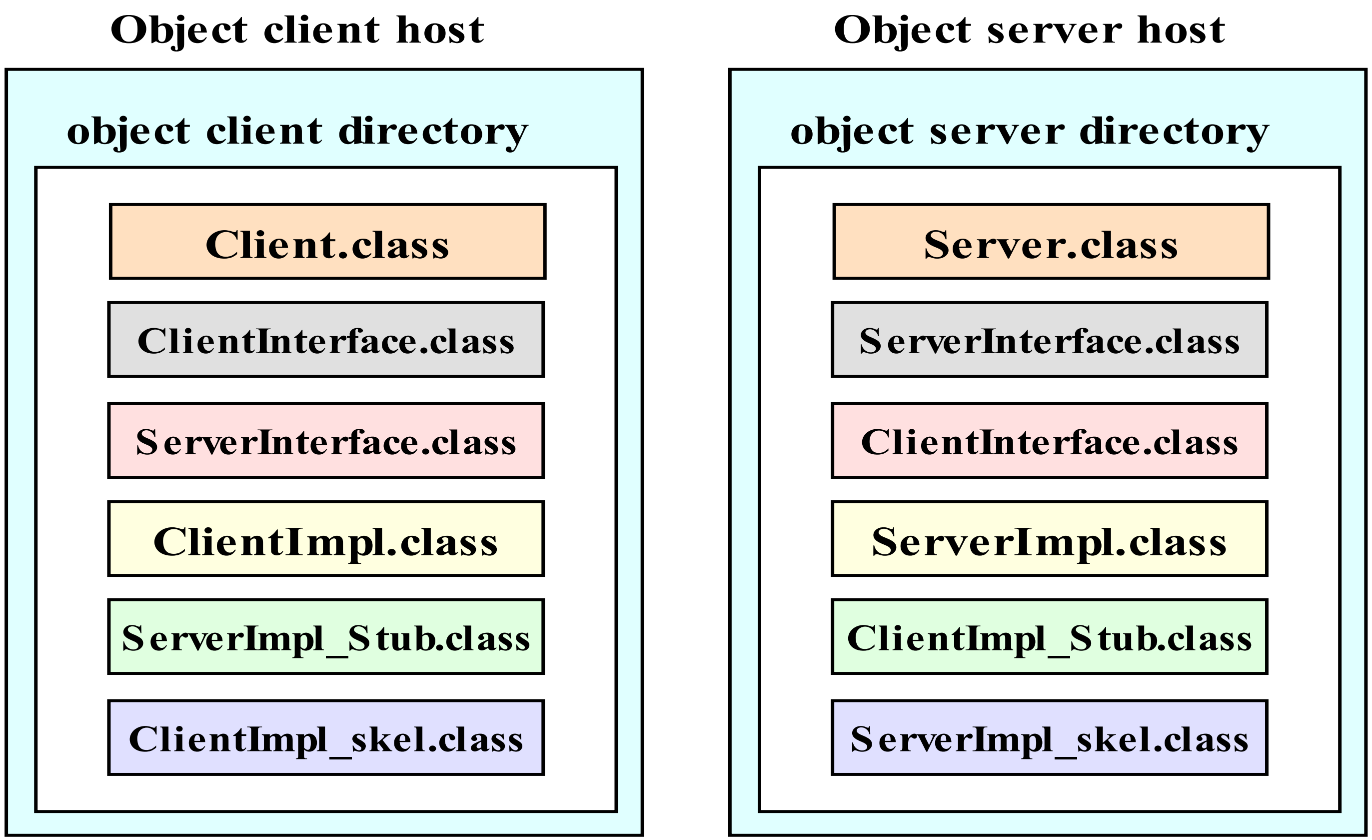


Callback Client-Server Interactions

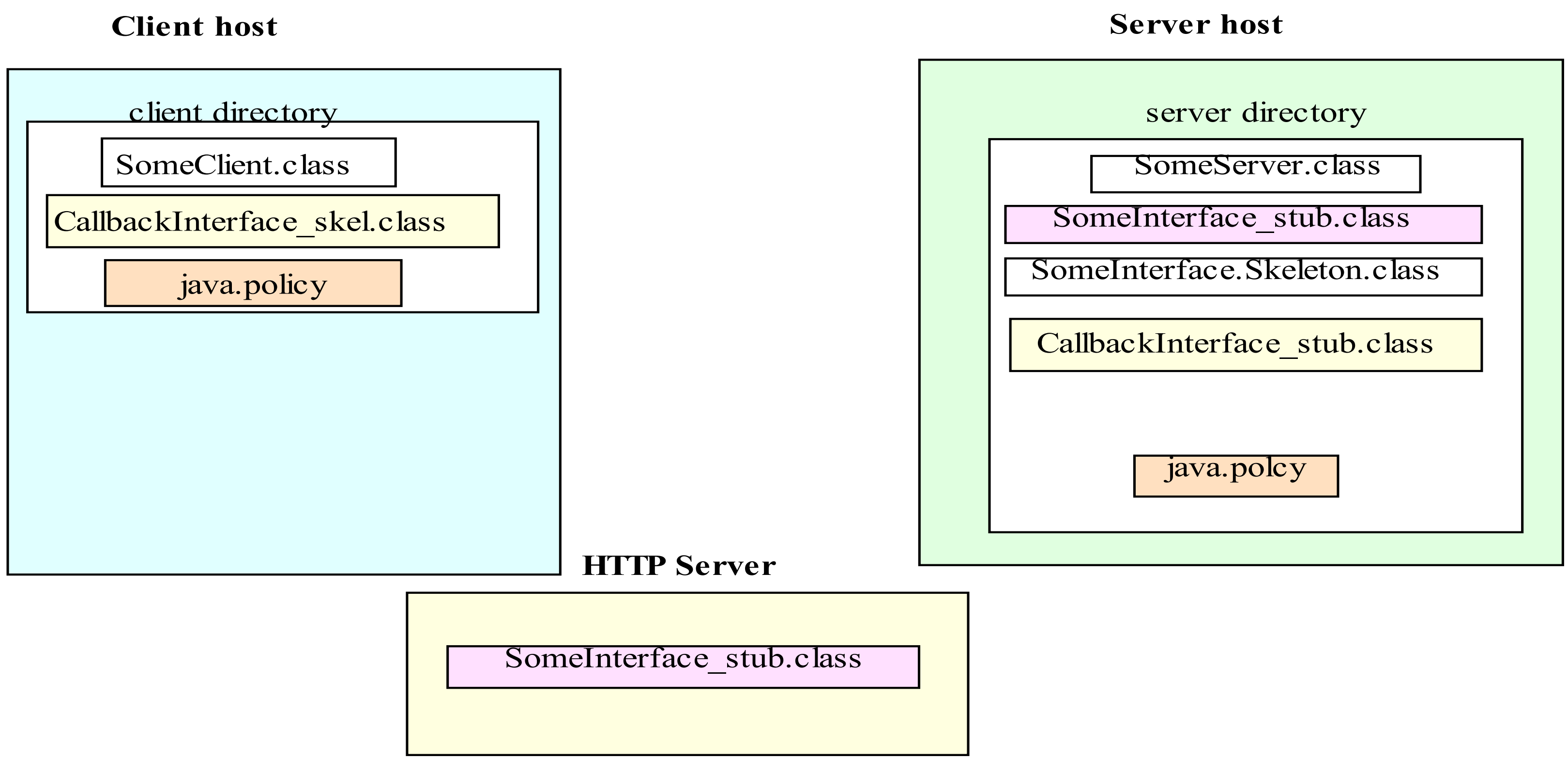


1. Client looks up the interface object in the RMIregistry on the server host.
2. The RMIRegistry returns a remote reference to the interface object.
3. Via the server stub, the client process invokes a remote method to register itself for callback, passing a remote reference to itself to the server. The server saves the reference in its callback list.
4. Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the interface object.
5. When the anticipated event takes place, the server makes a callback to each registered client via the callback interface stub on the server side and the callback interface skeleton on the client side.

Callback application files



RMI Callback file placements



RMI Callback Interface

- The server provides a remote method which allows a client to register itself for callbacks.
- A Remote interface for the callback is needed, in addition to the server-side interface.
- The interface specifies a method for accepting a callback from the server.
- The client program is a subclass of RemoteObject and implements the callback interface, including the callback method.
- The client registers itself for callback in its main method.
- The server invokes the client’s remote method upon the occurrence of the anticipated event.

Callback Client Interface

```
import java.rmi.*;

/* This is a remote interface for illustrating RMI client callback.  */

public interface CallbackClientInterface extends java.rmi.Remote{

// This remote method is invoked by a callback server to make a callback
// to an client which implements this interface.
// @param message - a string containing information for the client
// to process upon being called back. public String

 notifyMe(String message) throws java.rmi.RemoteException; }

// end interface
```

Callback Client Implementation

```
import java.rmi.*;
import java.rmi.server.*;
/* This class implements the remote interface
    CallbackClientInterface. */
public class CallbackClientImpl extends UnicastRemoteObject
    implements CallbackClientInterface {
    public CallbackClientImpl() throws RemoteException {
            super( ); }
    public String notifyMe(String message){
            String returnMessage = "Call back received: " + message;
            System.out.println(returnMessage); return returnMessage;
    }
}
// end CallbackClientImpl class
```

Callback Server Interface

```
import java.rmi.*;
/* This is a remote interface for illustrating RMI client callback. */
public interface CallbackServerInterface extends Remote {
    public String sayHello( ) throws java.rmi.RemoteException;
    // This remote method allows an object client to register for callback
    // @param callbackClientObject is a reference to the object of the
    // client; to be used by the server to make its callbacks.
    public void registerForCallback( CallbackClientInterface
          callbackClientObject ) throws java.rmi.RemoteException;
    // This remote method allows an object client to cancel its registration
    // for callback
    public void unregisterForCallback( CallbackClientInterface
          callbackClientObject) throws java.rmi.RemoteException; }
```